Large-Scale Behavioral Targeting

Ye Chen*
eBay Inc.
2145 Hamilton Ave
San Jose, CA 95125
yechen1@ebay.com

Dmitry Pavlov†
Yandex Labs
330 Primrose Road, Suite 306
Burlingame, CA, 94010
dmitry-pavlov@yandex-team.ru

John F. Canny
Computer Science Division
University of California
Berkeley, CA 94720
jfc@cs.berkeley.edu

ABSTRACT

Behavioral targeting (BT) leverages historical user behavior to select the ads most relevant to users to display. The state-of-the-art of BT derives a linear Poisson regression model from fine-grained user behavioral data and predicts click-through rate (CTR) from user history. We designed and implemented a highly scalable and efficient solution to BT using Hadoop MapReduce framework. With our parallel algorithm and the resulting system, we can build above 450 BT-category models from the entire Yahoo's user base within one day, the scale that one can not even imagine with prior systems. Moreover, our approach has yielded 20% CTR lift over the existing production system by leveraging the well-grounded probabilistic model fitted from a much larger training dataset.

Specifically, our major contributions include: (1) A MapReduce statistical learning algorithm and implementation that achieve optimal data parallelism, task parallelism, and load balance in spite of the typically skewed distribution of domain data. (2) An in-place feature vector generation algorithm with linear time complexity $O(n)$ regardless of the granularity of sliding target window. (3) An in-memory caching scheme that significantly reduces the number of disk IOs to make large-scale learning practical. (4) Highly efficient data structures and sparse representations of models and data to enable fast model updates. We believe that our work makes significant contributions to solving large-scale machine learning problems of industrial relevance in general. Finally, we report comprehensive experimental results, using industrial proprietary codebase and datasets.

Categories and Subject Descriptors

I.5.2 [**Design Methodology**]: Distributed Data Mining, High-Performance and Terascale Computing, Parallel Data Mining, Statistical Methods, User Modeling.

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Behavioral targeting, large-scale, grid computing

1. INTRODUCTION

Behavioral targeting (BT) leverages historical user behavior to select the most relevant ads to display. A well-grounded statistical model of BT predicts click-through rate (CTR) of an ad from user behavior, such as ad clicks and views, page views, search queries and clicks. Behavioral data is intrinsically in large scale (e.g., Yahoo! logged 9 terabytes of display ad data with about 500 billion entries[1] on August, 2008), albeit sparse; particularly ad click is a very rare event (e.g., the population CTR of automotive display ads is about 0.05%). Consequently, to fit a BT predictor with low generalization error requires vast amounts of data (e.g., our experiments showed that the generalization error monotonically decreased as the data size increased to cover all users). In practice, it is also desired to refresh models often and thus to constrain the running time of training, given the dynamic nature of user behavior and the volatility of ads and pages.

In this paper we present a scalable and efficient solution to behavioral targeting using Hadoop [1] MapReduce [9] framework. First, we introduce the background of BT for online display advertising. Second, we describe linear Poisson regression, a probabilistic model for count data such as online user behavioral data. We then focus on the design of such grid-based algorithms that can scale to the entire Yahoo! user base, with a special effort to share our experiences in addressing practical challenges during implementation and experiments. Finally, we show the experimental results on some industrial datasets, compared with the existing system in production. The contribution of this work includes the successful experiences at Yahoo! in parallelizing statistical machine learning algorithms to deal effectively with web-scale data using MapReduce framework, the theoretical and empirical insights into modeling very large user base.

*Work conducted at Yahoo! Labs, 701 First Ave, Sunnyvale, CA 94089.
†Work conducted at Yahoo! Labs, 701 First Ave, Sunnyvale, CA 94089.

[1]An entry of display ad data contains ad click, ad view and page view events conducted by a user (cookie) within a second.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'09, June 28–July 1, 2009, Paris, France.
Copyright 2009 ACM 978-1-60558-495-9/09/06 ...$5.00.

2. BACKGROUND

Behavioral targeting is yet another application of modern statistical machine learning methods to online advertising. But unlike other computational advertising techniques, BT does not primarily rely on contextual information such as query ("sponsored search") and web page ("content match"). Instead, BT learns from past user behavior, especially the implicit feedback (i.e., ad clicks) to match the best ads to users. This makes BT enjoy a broader applicability such as graphical display ads, or at least a valuable user dimension complementary to other contextual advertising techniques.

In today's practice, behaviorally targeted advertising inventory comes in the form of some kind of demand-driven taxonomy. Two hierarchical examples are "Finance, Investment" and "Technology, Consumer Electronics, Cellular Telephones". Within a category of interest, a BT model derives a relevance score for each user from past activity. Should the user appear online during a targeting time window, the ad serving system will qualify this user (to be shown an ad in this category) if the score is above a certain threshold. One *de facto* measure of relevance is CTR, and the threshold is predetermined in such a way that both a desired level of relevance (measured by the cumulative CTR of a collection of targeted users) and the volume of targeted ad impressions (also called reach) can be achieved. It is obvious that revenue is a function of CTR and reach.

3. LINEAR POISSON REGRESSION

We describe a linear Poisson regression model for behavioral count data [5]. The natural statistical model of count data is the Poisson distribution, and we adopt the linear mean parameterization. Specifically, let y be the observed count of a target event (i.e., ad click or view within a category), λ be the expected count or the mean parameter of the Poisson, $\mathbf{w}$ be the weight vector to be estimated, and $\mathbf{x}$ be the "bag-of-words" representation of feature event counts for a user. The probability density is:

$$p(y) = \frac{\lambda^y \exp(-\lambda)}{y!}, \text{where } \lambda = \mathbf{w}^\top \mathbf{x}. \tag{1}$$

Given a collection of user behavioral data $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where n is the number of training examples, we wish to find a weight vector $\mathbf{w}$ that maximizes the data log likelihood:

$$\ell = \sum_i \Big(y_i \log(\mathbf{w}^\top \mathbf{x}_i) - \mathbf{w}^\top \mathbf{x}_i - \log(y_i!) \Big), \tag{2}$$

where i denotes a user or a training example[2]. Taking the derivative of the log likelihood with respect to w_j yields:

$$\frac{\partial \ell}{\partial w_j} = \sum_i \left(\frac{y_i}{\lambda_i} x_{ij} - x_{ij} \right), \tag{3}$$

where w_j is the coefficient and x_{ij} is the regressor, respectively, of a feature indexed by j. The log likelihood function is globally concave; therefore the maximum likelihood estimator (MLE) of $\mathbf{w}$ can be uniquely found by gradient descent or, more efficiently, by the Newton-Raphson method. Better yet, we adapt a multiplicative recurrence proposed by Lee and Seung [11] to our optimization problem by constraining $\mathbf{w} \geq 0$. The multiplicative update rule is:

$$w_j' \leftarrow w_j \frac{\sum_i \frac{y_i}{\lambda_i} x_{ij}}{\sum_i x_{ij}}, \text{where } \lambda_i = \mathbf{w}^\top \mathbf{x}_i. \tag{4}$$

The assumption of non-negative weights is intuitive in that it imposes a positive effect of one-unit change in regressors (feature event counts) on the conditional mean (expected target event count), while yielding an efficient recurrence.

For predicting ad CTR of a user i within a target category k, we use the unbiased estimator constructed from the conditional Poisson means of clicks and views respectively, with Laplacian smoothing addressing data sparseness,

$$\widehat{\text{CTR}}_{ik} = \frac{\lambda_{ik}^{\text{click}} + \alpha}{\lambda_{ik}^{\text{view}} + \beta}, \tag{5}$$

where α and β are the smoothing constants that can be defined globally for all categories or specifically to each category. Note that α/β gives the default CTR of a "new user"[3] without any history, a natural choice is letting α/β be the cumulative CTR of all users in that category (also called population CTR). Since $\lambda_i = \mathbf{w}^\top \mathbf{x}_i$, the CTR prediction is inherently personalized (one Poisson mean for each user) and dynamic ($\mathbf{x}_i$ can be updated in real time as event stream comes in). The performance of a BT model is evaluated mainly via click-view ROC curve and, of greater business relevance, the CTR lift at a certain operating point of reach.

Finally, we comment on the choice of linear parameterization of the Poisson mean $\lambda = \mathbf{w}^\top \mathbf{x}$ (identity link function), instead of the canonical exponential form $\lambda = \exp(\mathbf{w}^\top \mathbf{x})$ (log link function). Recall that we assume $w_j \geq 0 \ \forall j$, the identity link is thus possible for Poisson since $\lambda \geq 0$. In model training, it can be readily shown that fitting a Poisson regression with the log link function involves computing $\exp(\mathbf{w}^\top \mathbf{x}_i) \ \forall i$ in each recurrence, instead of $\mathbf{w}^\top \mathbf{x}_i \ \forall i$ in the linear case. This slight efficiency gain by the latter can become non-trivial when model needs to be refreshed very often, while i may iterate over billions of examples. The rationale behind, however, comes more from the practical considerations in online prediction. Keep in mind that an online BT system needs to make ad serving decisions in real time (in the order of millisecond); thus it is generally impractical to score users from scratch (i.e., computing $\lambda = \mathbf{w}^\top \mathbf{x}$ from the entire history of $\mathbf{x}$). In practice, expected clicks and views for all users are computed offline *a priori* (e.g., daily) and updated incrementally online, usually with some form of decay. The simple linear form of the predictor makes incremental scoring possible by maintaining the additivity of the update function. More precisely, given a new event Δx_j of feature j taking place, the predictor can be updated as:

$$\lambda' = \lambda \delta^{\Delta t} + w_j \Delta x_j, \tag{6}$$

where λ' is the new estimator, λ is the previous one, and $\delta^{\Delta t}$ is an exponential decay with a factor of δ and Δt time elapsed. On the other hand, the exponential form increments the score as:

$$\lambda' = \exp\left[(\log \lambda) \delta^{\Delta t} + w_j \Delta x_j \right], \tag{7}$$

[2]Depending on the approach to generating training examples $(\mathbf{x}_i, y_i)$, one user could contribute to multiple examples.

[3]The "new user" phenomenon is not uncommon in BT, since in practice: (1) we can only track a finite amount of history; and (2) user is tracked by cookie, which is volatile.

where the logarithmic and exponential operations are extra computing cost.

4. LARGE-SCALE IMPLEMENTATION

In this section we describe the parallel algorithms of fitting the Poisson regression model using Hadoop MapReduce framework. Our focus, however, is to elaborate on various innovations in addressing practical challenges in large-scale learning [7].

4.1 Data Reduction and Information Loss

Most practical learning algorithms with web-scale data are IO-bound and particularly scan-bound. In the context of BT, one needs to preprocess 20-30 terabytes (TB) raw data feeds of display ads and searches (one month's worth). A good design should reduce data size at the earliest opportunity. Data reduction is achieved by projection, aggregation and merging. In the very first scan of raw data, only relevant fields are extracted. We next aggregate event counts of a user (identified by cookie) over a configurable period of time and then merge counts with a same composite key[4] (cookie, time) into a single entry. On the other hand, the data preprocessing step should have minimum information loss and redundancy. For example, the time span of aggregation shall satisfy the time resolution required for model training and evaluation (e.g., next one-minute prediction). This step neither decays counts nor categorizes ads, hence loosely coupled with specific modeling logics. A marginal redundancy is introduced by using (cookie, time) as key, given that examples only need to be identified at the cookie level. In practice, we found this is a good trade-off, since using the composite key reduces the length of an entry and hence scales to very long history user data by alleviating any restrictions on the buffer size of a record. After preprocessing, the data size is reduced to 2-3TB.

4.2 Feature Selection and Inverted Indexing

Behavioral data is heterogeneous and sparse. A data-driven approach is to use granular events as features, such as ad clicks, page views and search queries. With such a fine-grained feature space, feature selection is not only theoretically sound (to overcome the curse of dimensionality), but also practically critical; since for instance the dimensionality of search queries can be unbounded. We adopt a simple frequency-based feature selection method. It does so by counting entity frequency in terms of touching cookies and selecting most frequent entities into our feature space. An entity refers to the name (unique identifier) of an event (e.g., ad id or query). Entity is one level higher than feature since the latter is uniquely identified by a (feature type, entity) pair. For example, a same ad id can be used by an ad click or view feature; similarly, a query term may denote a search feature, an organic or sponsored result click feature. In the context of BT, we consider three types of entities: ad, page, and search. Thus the output of feature selection is three dictionaries (or vocabularies), which collectively (offset by feature types) define an inverted index of features.

We select features simply based on frequency instead of other more sophisticated information-theoretic measures such as mutual information. Not only does the simple frequency counting enjoy computational efficiency; but also because we found empirically frequency-based method works best for sparse data. The joint distribution term in mutual information will be dominated by frequent features anyway. And also, mutual information estimates can be very noisy in sparse data.

We found several design tricks worked very well in practice. First, frequency is counted in cookie rather than event occurrence. This effectively imposes a robot filtering mechanism by enforcing one vote for each cookie. Second, to select the most frequent entities we apply a predefined threshold instead of fixing top-N from a sorted frequency table. The threshold is probed *a priori* given a desired feature dimensionality. But once an optimal threshold is determined, it can be used consistently regardless of the size of training data. The thresholding approach is more statistically sound since more features (thus more model parameters) require more data to fit; and vice versa, more data favors more features in a more expressive model. Third, MapReduce framework enforces sorting on key for input to reducers, which is required for reducers to efficiently fetch relevant partitions. In many cases such as thresholding, however, the logic within reducer does not require input to be sorted. Sorting some data type can be expensive (e.g., arbitrarily long string as cookies). To avoid this unnecessary overhead, an optimization is to swap key and value in mapper, given that (1) key data type is sorting-expensive, while value data type is not; and (2) swapping key and value does not compromise the data needs and computing cost in reducer. The last MapReduce job of feature selection that hashes selected entities into maps (dictionaries) is an example of this optimization. Indeed, our implementation does even better. Once the frequency of an entity is summed, the threshold is applied immediately locally and in-memory; and hence the long tail of the frequency table is cut from the traffic to the last hashing step.

4.3 Feature Vector Generation in $\mathcal{O}(1n)$

For iterative algorithms for optimization commonly used in statistical learning, one needs to scan the training data multiple times. Even for a fast-convergent method such as the multiplicative recurrence in our case, the algorithm requires 15-20 passes of training data to converge the MLE of model parameters. Consequently, great efforts should be made to design a data structure optimized for sequential access, along both the user and feature dimensions; while materializing any data reduction and pre-computing opportunities. Behavioral count data is very sparse by nature, thus a sparse vector representation should also be used. Specifically, an example $(\mathbf{x}_i, \mathbf{y}_i)$ consists of two vectors, one for features $\mathbf{x}_i$ and the other for targets $\mathbf{y}_i$[5]. Each vector is represented as a pair of arrays of same length NNZ (number of nonzero entries); one of integer type for feature/target indices and the other of float type for values (float for possible decaying), with a same array index coupling an (index, value) pair. This data structure can be viewed as a flattened adaptation of Yale sparse matrix representation [10]. We further split the representations of features and targets for fast access of both, particularly the tensor product $\mathbf{y}_i \otimes \mathbf{x}_i$, the major computational cost for model updates.

[4]We use composite key here in a logical sense, while physically Hadoop only provides a simple key/value storage architecture. We implemented composite key as a single concatenated key.

[5]One example may contain multiple targets for different BT-category and click/view models.

With the inverted index built from feature selection, we reference a feature name by its index in generating examples and onwards. The original feature name can be of any data type with arbitrary length; after indexing the algorithm now efficiently deals with integer index consistently. Several pre-computations are carried out in this step. First, feature counts are further aggregated into a typically larger time window (e.g., one-month feature window and one-day target window); and target counts are aggregated from categorized feature counts. Second, decay counts exponentially over time to account for the freshness of user behavior. Third, realize causal or non-causal approaches to generating examples. The causal approach collects features before targets temporally; while the non-causal approach generates targets and features from a same period of history. Although the causal method seems more intuitive and must be followed in evaluation, the non-causal way has advantages in increasing the number of effective examples for training and hence more suitable for short-term modeling. The data size now, one that will be directly consumed by weight initialization and updates, is 200-300 gigabytes (GB) with binary and compressed storage format.

It is generally intractable to use algorithms of time complexity higher than linear $O(n)$ in solving large-scale machine learning problems of industrial relevance. Moreover, unlike traditional complexity analysis, the scalar c of a linear complexity $O(cn)$ must be seriously taken into account when n is easily in the order of billion. BT is a problem of such scale. Our goal in time complexity is $O(1n)$, where n is the number of examples keyed on (cookie, time). Recall that ad click is a very rare event, while it is a target event thus carrying arguably the most valuable information in predicting CTR. The size of a sliding target window should be relatively small for the following reasons. Empirically, a large window (e.g., one-week) effectively discards many (feature, target) co-occurrences given that a typical user session lasts less than one hour. Theoretically, for a large window and hence large target event counts, the assumed Poisson approaches a Gaussian with a same mean and may suffer from overdispersion [3]. In online prediction on the other hand, one typically estimates target counts in a window of several minutes (time interval between successive ad servings). Suppose that the number of sliding windows is t, a naïve algorithm would scan the data t times and thus have a complexity of $O(tn)$. When t increases, $O(tn)$ becomes unacceptable. For example, per-minute sliding over one week for short-term modeling gives $10,080$ windows.

We develop an algorithm of $O(1n)$ smoothed complexity [12] for generating examples, regardless of the number of sliding windows t. The essence of the algorithm is the following: (1) cache in memory all inputs of a cookie; (2) sort events by time and hence forming an event stream; (3) precompute the boundaries of feature/target sliding windows; (4) maintain three iterators on the event stream, referencing previous featureBegin, current featureBegin and targetBegin, respectively; (5) use one pair of objects (e.g., TreeMap in Java) to respectively hold the feature and target vectors, but share the object pair for all examples. As the feature and target windows slide forward, advance the iterators accordingly to generate an example for the current window using in-place increment, decrement, and decay. In the worst case, one scan of each of the three iterators is sufficient for generating all examples for the cookie in question. In a typical causal setup, the target window is much smaller than the feature window; hence the smoothed complexity is $O(1n)$. The formalism and schematic of the algorithm are shown in Algorithm 1 and Figure 1, respectively. Note that we let FeatureVector collectively denote the 2-tuple of input feature and target feature vectors.

Algorithm 1: Feature vector generation

```
/* We denote a datum in MapReduce as ⟨key, value⟩,
   use ':'  as field delimiter within key or value,
   and '...'  for repetition of foregoing fields.  */
Data structure: FeatureVector
begin
    /* Array notation:  dataType[arrayLength]      */
    int[targetLength] targetIndexArray;
    float[targetLengh] targetValueArray;
    int[inputLength] inputIndexArray;
    float[inputLength] inputValueArray;
end
  Input: ⟨cookie:timePeriod,
         featureType:featureName:featureCount...⟩
  Output: ⟨cookieIndex, FeatureVector⟩
MapReduce: PoissonFeatureVector;
Mapper → ⟨cookie,
timePeriod:featureType:featureName:featureCount...⟩;
Reducer → ⟨cookieIndex, FeatureVector⟩;
begin
    /* For a cookie                                */
    compute boundaries of t pairs of feature/target windows;
    cache events and sort values by timePeriod;
    initialize iterators and TreeMaps;
    /* Slide window forward                        */
    for i ← 1 to t do
        advance prevFeatureBegin to decrement feature
        counts in-place;
        decay feature counts incrementally;
        advance currFeatureBegin to increment feature
        counts in-place;
        advance currTargetBegin to increment target counts;
        robot filtering and stratified sampling;
        bookkeeping reducer-local total counts;
        output FeatureVector;
    end
end
```

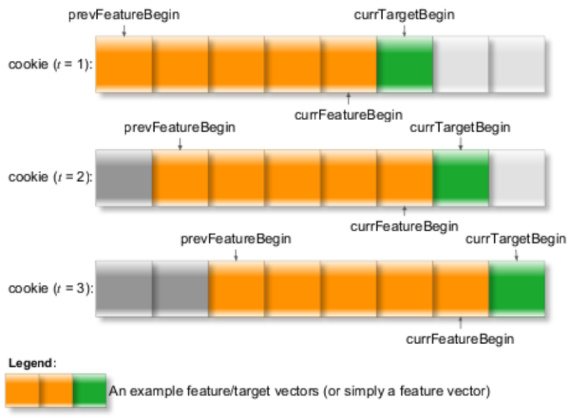

Figure 1: Feature vector generation in $\mathcal{O}(1n)$

4.4 Data-driven Weight Initialization

Model initialization involves assigning initial weights (coefficients of regressors) by scanning the training set D once. To exploit the sparseness of the problem, one shall use some data-driven approach instead of simply uniformly or randomly assigning weights to all parameters, as many gradient-based algorithms do. A data-driven initialization will drastically speed up weights convergence since, for example, under the multiplicative update rule as Eq. (4) those weights with no data support will remain zeros as initialized. We define a unigram(j) as one occurrence of feature j, and a bigram(k, j) as one co-occurrence of target k and feature j. The basic idea is to allocate the weight w_{kj} as normalized co-occurrences of (k, j), i.e., a bigram-based initialization. Here normalization can be performed per example through its total feature counts, and globally through unigram and/or bigram counts. We implement two weight initialization methods under different motivations. The first method uses feature-specific normalization by total feature unigram counts over all examples, motivated by the idea of tf-idf,

$$w_{kj} \leftarrow \frac{\sum_i \frac{y_{ik}x_{ij}}{\sum_{j'} x_{ij'}}}{\sum_i x_{ij}}. \tag{8}$$

The second method uses target-specific normalizer involving total unigram and bigram counts, motivated by the highly skewed distribution of traffic over categories,

$$w_{kj} \leftarrow \frac{\sum_i (y_{ik}x_{ij}) \sum_i y_{ik}}{\sum_{j'} \left[\sum_i (y_{ik}x_{ij'}) \sum_i x_{ij'}\right]}. \tag{9}$$

4.5 Parallel Multiplicative Recurrence

We wish to estimate the MLE of a dense weight matrix from a sparse data matrix D. We adopt a highly effective multiplicative update rule arising from non-negative matrix factorization (NMF) [11], given that D contains count data and weights are assumed to be non-negative. More precisely, let W be a $d \times m$ weight matrix where d is the number of targets and m is the number of features. A dense view of D is a $n \times (d + m)$ matrix which can be further blocked into a $n \times d$ target counts matrix Y and a $n \times m$ feature counts matrix X, i.e., $D = [Y\ X]$. The MLE of W can be regarded as the solution to an NMF problem $Y^\top \approx WX^\top$ where the quality of factorization is measured by data log likelihood [4]. Since both Y and X are given, we directly apply the multiplicative algorithm in [11] to compute W thus yielding our recurrence in Eq. (4).

The computational performance of the multiplicative update in Eq. (4) is dominated by counting bigrams (per-example normalized as in the numerator of the multiplicative factor), while the global normalizing unigram counts (the denominator of the multiplicative factor) are pre-computed in feature vector generation. Iterative learning algorithms typically encounter parallelization bottleneck in synchronizing model parameters after each iteration [6]. In solving Poisson MLE in our case, for each iteration the final multiplicative update of the weight vector $\mathbf{w}_k$ of a target variable k has to be carried out in a single node to output this weight vector. Notice that the number of targets d can be arbitrarily small; and the traffic distribution over targets (categories) is by nature highly skewed. Consequently, an unwise parallel design would suffer from suboptimal task parallelism and poor load balance. Our algorithm successfully addresses the above parallelization challenges as follows.

4.5.1 Scalable Data Structures

To achieve optimal task parallelism, we represent the weight matrix W as d dense vectors (arrays) of length m, each $\mathbf{w}_k$ for a target variable k. First, using weight vectors is more scalable in terms of memory footprint than matrix representation. Assume that $d = 500$ and $m = 200,000$, a dense W in float requires 400 megabytes (MB) memory. Reading the entire matrix in memory, as one previous standalone implementation does, is unscalable for clusters of commodity machines. For example, in Yahoo's Hadoop clusters, each node has 8GB RAM which is typically shared by 8 JVM processes and hence 1GB per JVM. The vector representation scales in both target and feature dimensions. A weight vector is read in memory on demand and once at a time; and hence d can be arbitrarily large. The memory footprint of a vector becomes bounded, e.g., a 200MB RAM can hold a vector of 50 million float weights. A three-month worth of Yahoo's behavioral data without feature selection contains features well below 10 million. Second, the weight vector data structure facilitates task parallelism since a node only needs to retrieve those vectors relevant to the data being processed. Third, the dense representation of $\mathbf{w}_k$ makes the dot product $\lambda_{ik} = \mathbf{w}_k^\top \mathbf{x}_i$ very efficient. Recall that feature vector uses sparse array data structure. Given the relevant $\mathbf{w}_k$ as a dense array in memory, one loop of the $\mathbf{x}_i$ sparse array is sufficient for computing the dot product, with a smoothed complexity of $O(m_\mathbf{x})$ where $m_\mathbf{x}$ is the typical NNZ in $\mathbf{x}_i$. A dot product of two sparse vectors of high dimensionality is generally inefficient since random access is not in constant time as in dense array. Even a sort-merge implementation would only yield a complexity of $O(m_\mathbf{w})$ where $m_\mathbf{w}$ is the typical NNZ in $\mathbf{w}_k$ and $m_\mathbf{x} \ll m_\mathbf{w} < m$. The choice of sparse representation for feature vector is thus readily justified by its much higher sparseness than weight vector[6] and the even higher dimensionality of n.

4.5.2 Fine-grained Parallelization

For updating the weight matrix $W = [w_{kj}]_{d \times m}$ iteratively, we distribute the computation of counting bigrams by the composite key (k, j) which defines an entry w_{kj} in W. A naïve alternative is distributing either rows by k or columns by j; both however suffer from typically unbalanced traffics (some k or j dominates the running time) and the overhead of synchronizing bigram(k, j). By distributing (k, j), the algorithm yields an optimal parallelization independent of the characteristics of domain data, with no application-level parallelization needed. Distributing composite keys (k, j) effectively pre-computes total bigram counts of all examples in a fully parallel fashion before synchronizing weight vectors; and thus making the last synchronization step as computationally light-weighted as possible. This, indeed, is the key to a successful parallel implementation of iterative learning algorithms. In our implementation, the weights synchronization along with update only takes less than two minutes. Recall that MapReduce framework only provides a single-key storage architecture. In order to distribute (k, j) keys, we need an efficient function to construct a one-value composite key from two simple keys and to recover the sim-

[6]Define the sparseness of a vector as the percentage of zero entries, denoted as $\eta(\mathbf{x})$. For Yahoo's behavioral dataset, the converged weight vectors have an average $\eta(\mathbf{w}_k) = 29.7\%$ and 84% $\mathbf{w}_k$'s has an $\eta(\mathbf{w}_k) \leq 50\%$; while for feature vectors it's almost for certain that $\eta(\mathbf{x}_i) > 97\%$.

ple keys back when needed. Specifically, we define the following operators for this purpose: (1) bigramKey(k, j) = a long integer obtained by bitwise left-shift 32-bit of k and then bitwise OR by j; (2) k = an integer obtained from the high-order 32-bit of bigramKey(k, j); (3) j = an integer obtained from the low-order 32-bit of bigramKey(k, j).

4.5.3 In-memory Caching

The dense weight vector representation is highly scalable, but raises challenges in disk IO. Consider a naïve implementation that reads weight vectors from disk on demand as it sequentially processes examples. Suppose that there are n examples, d targets, and on average each example contains $d_{\mathbf{x}}$ targets. File IO generally dominates the running time of large-scale computing. In the worst case of $d_{\mathbf{x}} = d$, the naïve algorithm thus has a complexity of $O(dn)$, which obviously is of no use in practice. We tackle this problem via in-memory caching. Caching weight vectors is, however, not the solution; since a small subset of examples will require all weight vectors sit in memory. The trick is to cache input examples. Now suppose that there are l caches for the n examples. After reading each cache into memory, the algorithm maintains a hash map of (target index, array index). This hash map effectively records all relevant targets for the cached examples, and meanwhile provides constant-time lookup from target index to array index to retrieve target counts. In the worst case of all caches hitting d targets, our algorithm yields a complexity of $O(dl)$, where $l \ll n$. We argue that caching input data is generally a very sound strategy for grid-based framework. For example, a Hadoop cluster of 2,000 nodes can distribute 256GB data into 128MB blocks with each node processing only one block on average, and thus $l = 1$ to 2. In-memory caching is also applied to the output of the first mapper that emits (bigramKey(k, j), bigram(k, j)) pairs for each example i, while aggregating bigram counts into a same bigram key for each cache. This output caching reduces disk writes and network traffic, similar to the function of combiner; while leveraging data locality in memory proactively.

The parallel algorithm, data structures, and in-memory caching for multiplicative update are also applied to model initialization. Notice that the multiplicative factor in Eq. (4) has an identical form as the first initialization method in Eq. (8), except that the per-example normalizer becomes the expected target counts instead of the total feature unigram counts. We show our parallel design formally in Algorithm 2, and schematically in Figure 2.

Algorithm 2: Parallel multiplicative recurrence

Input: ⟨cookieIndex, FeatureVector⟩
Output: updated $\mathbf{w}_k, \forall k$

MapReduce 1: PoissonMultBigram;
Function: `bigramKey`(k, j)
begin
  return *a long by bitwise-left-shift 32-bit k and bitwise-OR j*;
end
Function: `cacheOutput`$(key, value)$
begin
  if $outputCacheSize \geq upperBound$ **then**
    output and clear current cache;
  end
  cache and aggregate bigrams;
end
Function: `processInputCache()`
begin
  foreach k **do**
    read $\mathbf{w}_k$;
    foreach $\mathbf{x}_i \in inputCache$ **do**
      $\lambda_{ik} \leftarrow \mathbf{w}_k^\top \mathbf{x}_i$;
      $y_{ik} \leftarrow y_{ik}/\lambda_{ik}$;
      `cacheOutput(bigramKey`(k, j)`,`$y_{ik}x_{ij}$`)`, $\forall k, j$;
    end
  end
end
Function: `cacheInput`$(value)$
begin
  if $inputCacheSize \geq upperBound$ **then**
    `processInputCache()`;
    clear input cache;
  end
  cache bigrams and hash map of ⟨targetIndex, targetArrayIndex⟩;
  randomized feature/target partitioning if specified;
end
Mapper → ⟨`bigramKey`(k, j), $y_{ik}x_{ij}$⟩;
begin
  `cacheInput`$(FeatureVector)$;
end
Combiner: ValueAggregatorCombiner;
Reducer: ValueAggregatorReducer;
MapReduce 2: PoissonMultWeight;
Mapper: IdentityMapper;
Partitioner: by k (routing entries with a same target to a single reducer);
Reducer → $\mathbf{w}_k$;
begin
  $w'_{kj} \leftarrow w_{kj} \frac{\sum_i (y_{ik}x_{ij}/\lambda_{ik})}{\sum_i x_{ij}}$;
  L^1-norm or L^2-norm regularization if specified;
end

5. EXPERIMENTS

5.1 Dataset and Parameters

We conducted a comprehensive set of large-scale experiments using the enormous Yahoo's user behavioral data, to evaluate the prediction accuracy and scalability of our parallel Poisson regression model. In each experiment reported below, the controlled parameters are the ones we found empirically superior, as follows. The training data was collected from a 5-week period of time (2008-09-30 to 2008-11-03) where the first four weeks formed the explanatory variables $\mathbf{x}$ and the last week was for generating the response variable y. We used all 512 buckets[7] of user data, which gave above 500 millions training examples and approximately 3TB preprocessed and compressed data. The training examples were generated in a causal fashion; with a target window of size one-day, sliding over a one-week period, and preceded by a 4-week feature window (also sliding along with the target window). We leveraged six types of features: ad clicks and views (sharing a same dictionary of ads), page views (from a dictionary of pages), search queries, algorithmic and sponsored result clicks (sharing a same dictionary of queries). For feature selection, we set the frequency thresholds in terms

[7]A bucket is a random partition of cookies, where the partitioning is done by a hash function of cookie string.

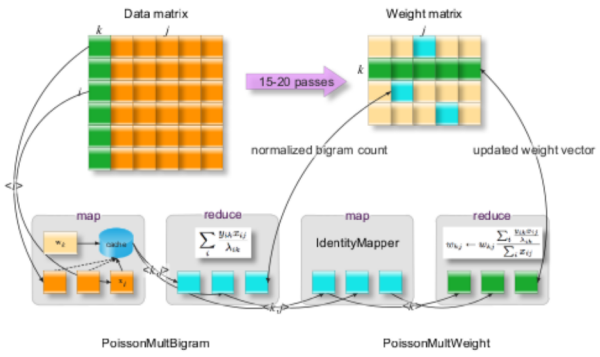

Figure 2: Parallel multiplicative recurrence

of touching cookies to be: 50,000 for ads, 10,000 for pages, and 20,000 for queries. This gave about 150K features comprised of 40K ads (×2), 40K pages, and 10K queries (×3). For robot filtering, we removed examples with the number of distinct events above 5,000. After model initialization using the second method as described in Section 4.4, we performed 17 iterations of multiplicative updates to converge weights. Model evaluation was done using 32 buckets of data (a 1/16 sample) from the next day (2008-11-04) following the training period. To simulate online prediction, we set the exponential decay ratio $\delta = \sqrt[14\text{-day}]{1/2}$ (i.e., a half-life of 14-day); and a 6-minute latency between a 5-week feature window and a 6-minute target window, sliding over one day.

The prediction accuracy is measured by two metrics: (1) the relative CTR lift over a baseline at a certain operating point of view recall (also called reach in ad targeting terminology), where the CTR is normalized by the population CTR to eliminate potential variances across buckets and time; and (2) the area under the click-view ROC curve. A click-view ROC curve plots the click recall vs. the view recall, from the testing examples ranked in descending order of predicted CTR. Each point on the ROC curve gives the precision in a relative sense (click recall/view recall) corresponding to a particular view recall. The higher the area under the ROC curve, the more accurate the predictor; and a random predictor would give a ROC area of 0.5 (a diagonal ROC). Since we built models for 60 major BT categories in one batch, we report average CTR lift weighted by views in each category, and average ROC area weighted by clicks in each category. The scalability of our algorithm is measured by the increase in running time (in hours) with respect to input size. All our experiments were run on a 500-node Hadoop cluster of commodity machines (2× Quad Core 2GHz CPU and 8GB RAM).

5.2 The Baseline Model

We compared our results with a baseline model using linear regression, which is a prior solution to BT [8]. The linear regression model has two groups of covariates: intensity and recency. Intensity is an aggregated event count, possibly with decay; while recency is the time elapsed since a user had a event most recently. Both covariate groups contain the same six event types as in our Poisson model; but counts are aggregated into the category being modeled and restricted to that category. The response variable is a binary variable indicating clicking ($y = 1$) or not ($y = 0$); thus the regression model predicts the propensity for clicking (similar as CTR but unbounded). The linear regression model was trained with quadratic loss; but constrained on the signs of coefficients based on domain knowledge, i.e., all intensity coefficients are non-negative except for ad views and all recency coefficients are non-positive except for ad views. The model has two components: a long-term model was trained to predict the click propensity for the next day using at least one-month worth of user history; and a short-term model was trained to predict the click propensity in the next hour using at least one-week worth of data. The final clickability score is the product of the long-term and short-term scores (the independence assumption).

5.3 Results

5.3.1 *Data Size*

Recall that ad click is a very rare event while carries probably the most important user feedback information. The feature space of granular events, on the other hand, has an extremely high dimensionality. It is therefore desirable to use more data for training. One major objective of our large-scale implementation is to scale up to the entire Yahoo's user data. To evaluate the effect of the size of training data on the prediction and computational performances, we varied input data size from 32 buckets to 512 buckets. The results show, as in Table 1, that as the data size increases, the prediction accuracy increases monotonically, while the run-time grows sub-linearly.

Table 1: The Effect of Training Data Size

# Buckets	32	64	128	256	512
CTR lift	0.1583	0.2003	0.2287	0.2482	0.2598
ROC area	0.8193	0.8216	0.8234	0.8253	0.8267
Run-time	2.95	3.78	6.95	7.43	14.07

One prior implementation contains a nonparallel training routine, and trivially parallelized (data parallelism only) feature vector generation and evaluation routines. For the same batch of 60 BT-category models trained on 256 buckets of data, 50K features, and with only one target window of size one-week (non-sliding), the running time was 29 hours. The majority of the time (over 85%) was spent on the nonparallel weight initialization and updates restricted to a single machine, while feature vector generation and evaluation were distributed across about 100 nodes using Torque and Moab. It only took our fully parallel implementation 7.43 hours, a 4× speed-up; even with 150K features and daily sliding target window. It is important to note that the prior implementation was not able to handle as large feature space or sliding window in tractable time primarily because of scalability limitations.

5.3.2 *Feature Selection*

When abundant data is available, a high-dimensional feature space may yield better model expressiveness. But as the number of features keeps increasing, the model becomes

overfitting. In practice, to find the optimal number of features is largely an empirical effort. This experiment reflects such an effort. We examined different numbers and combinations of features, as summarized in Table 2; and the results are shown in Table 3.

Table 2: The Parameters of Feature Selection

Total number	Ads (×2)	Pages	Queries (×3)
60K	10K	10K	10K
90K	20K	20K	10K
150K	40K	40K	10K
270K	80K	80K	10K
1.2M	100K	1M	10K

Table 3: The Effect of Feature Dimensionality

# Features	60K	90K	150K	270K	1.2M
CTR lift	0.2197	0.2420	0.2598	0.2584	0.2527
ROC area	0.8257	0.8258	0.8267	0.8267	0.8261
Run-time	14.87	13.52	14.07	13.08	16.42

The results show that 150K is the empirically optimal number of features given other parameters controlled as described in Section 5.1. This optimum is primarily a function of the size of training data. A similar study was performed on a 64-bucket training set using the prior nonparallel solution discussed in Section 5.3.1; and we found that 50K features was the optimal point for a 1/8 sample. As shown in the column of queries in Table 2, we controlled the number of queries unchanged in this experiment. This is because we found, from a prior study, that the contribution of query features to CTR prediction is insignificant relative to ads and pages. The run-time results shown in Table 3 confirm that the running time is approximately a constant w.r.t. the dimensionality of feature space. This suggests that our implementation is scalable along the feature dimension, which was made possible by in-memory caching input examples, reading weight vectors on-demand, and computing updates in batch, as discussed in Section 4.5.

5.3.3 Feature Vector Generation

As explained in Section 4.3, one key to a scalable solution to BT is a linear-time algorithm for feature vector generation. We developed such an algorithm by in-place incrementing and decrementing a shared map data structure; and hence typically one scan of the input data suffices for generating all examples. In this experiment, we verified the scalability of the feature vector generation routine, and the prediction performances resulted from different sizes of sliding target window. Over a one-week target period, we generated examples with a sliding target window of sizes 15-minute, one-hour, one-day, and one-week, respectively. The results are illustrated in Table 4.

The results show that, as the target window size reduces from one-week to 15-minute, the run-time for feature vector generation remains approximately constant; even though the number of active examples increases by 13 folds at the high end relative to the low. Here an active example is defined as the one having at least one ad click or view in any category being modeled. The total run-time does increase since the downstream modeling routines need to process more examples, but at a much lower rate than that of the number of examples increasing. As for prediction accuracy, one-day sliding gives the best CTR lift and ROC area.

Table 4: Linear-time Feature Vector Generation

Size of tgt. win.	15-min	1-hour	1-day	1-week
CTR lift	0.1829	0.2266	0.2598	−0.0086
ROC area	0.8031	0.8145	0.8267	0.7858
Act. ex. (10^6)	2,176	1,469	535	158
Run-time (fv-gen)	1.5	1.57	1.43	1.38
Run-time (total)	31.03	27.37	14.07	9.23

5.3.4 Stratified Sampling

The training examples can be categorized into three groups: (1) the ones with ad clicks (in any BT-categoy being modeled), (2) the ones with zero ad clicks but nonzero ad views, and (3) the ones with neither ad clicks nor views, or so-called negative examples. It is plausible that the first group carries the most valuable information for predicting CTR, followed by the second group and then the third. It has computational advantages to sample less important examples. In this experiment, we tested different stratified sampling schemes, where all nonzero-click examples were kept, view-only and negative examples were sampled independently at different rates. The results are summarized in Table 5.

Table 5: Stratified Sampling

Sampling rates	CTR lift	ROC area	Run-time
$\mathbf{neg} = 0; \mathbf{view} = 1$	0.2598	0.8267	14.07
$\mathbf{neg} = 0.2; \mathbf{view} = 1$	0.2735	0.8243	12.77
$\mathbf{neg} = 0.5; \mathbf{view} = 1$	0.2612	0.8208	13
$\mathbf{neg} = 1; \mathbf{view} = 1$	0.2438	0.8162	11.88
$\mathbf{neg} = 0; \mathbf{view} = 0.5$	0.2579	0.8280	8.9
$\mathbf{neg} = 0; \mathbf{view} = 0.2$	0.2462	0.8266	7.57
$\mathbf{neg} = 0; \mathbf{view} = 0$	−0.0328	0.7736	5.38

The negative examples only impact the denominator in the update formula as in Eq. (4). Since the denominator does not depend on λ_i, it can be pre-computed as a normalizer in the multiplicative factor; and then the multiplicative recurrence only needs to iterate over the active examples. Our implementation exploits this sparseness, thus the run-time is only sensitive to the view-only sampling rate. Table 5 shows that a small sampling rate for negative examples (0 or 0.2) combined with a large view-only sampling rate (0.5 or 1) yields superior results, which confirms the argument about different information contents in sub-populations.

5.3.5 Latency

In the offline evaluations reported on so far, we placed a 6-minute latency (also called gap) window between a 5-week feature window and a 6-minute target window. Assuming a uniform distribution of a target event over the target window, the expected latency was 9 minute. In other words, we disregarded any event happening during the 9-minute latency window for predicting that particular target event. This was to simulate an online prediction environment where the data pipeline was not real-time after an user event was triggered and before the production scoring system saw that event. However, user activities within the session where an ad is served, especially some task-based information, are

considered very relevant [2]. The objective of this experiment is to validate the potential of latency removal. The models were trained in the same way as described in Section 5.1, but evaluated with no gap and a one-minute sliding target window. As the results show, in Table 6, the latency-reduced evaluation yields a significantly higher prediction accuracy than the non real-time setup, by a 17% improvement in CTR lift and a 1.5% edge in ROC area. The ROC curves for the category "Technology" before and after latency removal are plotted in Figure 3.

Table 6: The Effect of Latency Removal

Latency	6-min gap 6-min target	no-gap 1-min target
CTR lift	0.2598	0.4295
ROC area	0.8267	0.8413

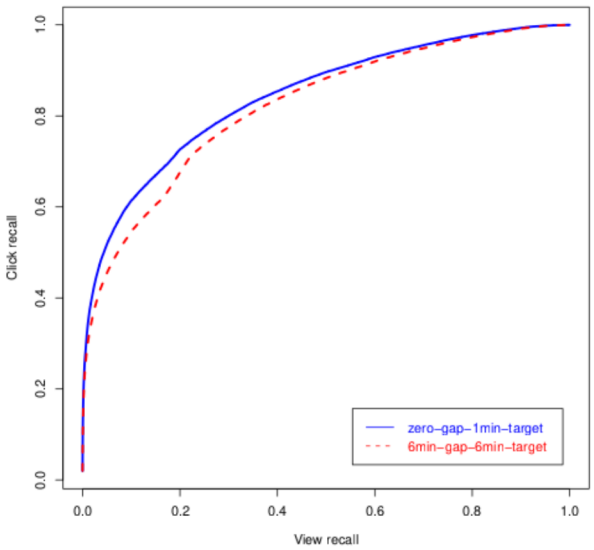

Figure 3: ROC plots before and after latency removal for the "Technology" category

6. DISCUSSION

Behavioral targeting is intrinsically a large-scale machine learning problem from the following perspectives. (1) To fit a BT predictive model with low generalization error and a desired level of statistical confidence requires massive behavioral data, given the sparseness the problem. (2) The dimensionality of feature space for the state-of-the-art BT model is very high. The linear Poisson regression model uses granular events (e.g., individual ad clicks and search queries) as features, with a dimensionality ranging from several hundred thousand to several million. (3) The number of BT models to be built is large. There are over 450 BT-category models for browser and login cookies need to be trained on a regular basis. Furthermore, the solution to training BT models has to be very efficient, because: (1) user interests and behavioral patterns change over time; and (2) cookies and features (e.g., ads and pages) are volatile objects.

Our grid-based solution to BT successfully addresses the above challenges through a truly scalable, efficient and flexible design and implementation. For example, the existing standalone modeling system could only manage to train 60 BT-category models using about one week end-to-end time. Our solution can build over 450 BT models within one day. The scalability achieved further allows for frequent model refreshes and short-term modeling. Finally, scientific experimentation and breakthroughs in BT requires such a scalable and flexible platform to enable a high speed of innovation.

7. REFERENCES

[1] http://hadoop.apache.org/.

[2] S. Agarwal, P. Renaker, and A. Smith. Determining ad targeting information and/or ad creative information using past search queries. *U.S. Patent 10/813,925*, filed: Mar 31, 2004.

[3] A. C. Cameron and P. K. Trivedi. *Regression Analysis of Count Data*. Cambridge University Press, 1998.

[4] J. Canny. GaP: a factor model for discrete data. *ACM Conference on Information Retrieval (SIGIR 2004)*, pages 122–129, 2004.

[5] J. Canny, S. Zhong, S. Gaffney, C. Brower, P. Berkhin, and G. H. John. Granular data for behavioral targeting. *U.S. Patent Application 20090006363*.

[6] E. Chang. Scalable collaborative filtering algorithms for mining social networks. In *The NIPS 2008 Workshop on "Beyond Search: Computational Intelligence for the Web"*, 2008.

[7] Y. Chen, D. Pavlov, P. Berkhin, and J. Canny. Large-scale behavioral targeting for advertising over a network. *U.S. Patent Application 12/351,749*, filed: Jan 09, 2009.

[8] C. Y. Chung, J. M. Koran, L.-J. Lin, and H. Yin. Model for generating user profiles in a behavioral targeting system. *U.S. Patent 11/394,374*, filed: Mar 29, 2006.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] N. E. Gibbs, W. G. Poole, Jr., and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 2(3):322–330, 1976.

[11] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. *Advances in Neural Information Processing Systems (NIPS)*, 13:556–562, 2000.

[12] D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *Journal of the ACM*, 51(3), 2004.